Building a Native XML-DBMS as a Term Project in a Database Systems Course

Christoph Koch, Dan Olteanu, and Stefanie Scherzinger
Lehrstuhl für Informationssysteme
Universität des Saarlandes, Saarbrücken, Germany
{koch,olteanu,scherzin}@infosys.uni-sb.de

1. INTRODUCTION

This is to report on a database systems course the first author held in the summer semester of 2005 at Saarland University, Saarbrücken, Germany.

This course was an experiment in several respects. For one, we wanted to teach a systems course with a practical part in which students apply the material taught to build the core of a database management system. Such a systems building effort seems to be quite common in top-tier US universities, but it is rare in Europe. One main reason for this is that European curricula often require students to take many small courses per term. Students then cannot be required to invest the time necessary for such a systems-building effort into an individual course. In Saarbrücken, this fortunately does not apply and students are expected to take only about two main courses per term. (The database systems course in Saarbrücken is worth 9 points in the European course-credit transfer system ECTS, which corresponds to an estimated workload of 20 hours per week.)

Second we wanted our students to do something reasonably new to facilitate follow-up bachelor's and master's projects. So we decided to have them build the core of a native XML-DBMS, an active research area with a number of exciting problems. This would allow students to do something that may never have been done before: Indeed, to our knowledge, no database systems course has had students build a native XML-DBMS before. (We are aware of a compiler construction course at UC San Diego in which a main-memory XQuery system was built, however [6].)

A main goal of the course was to convince the participants that systems research goes far beyond good programming. Students should get the opportunity to experience success in speeding up query evaluation by several orders of magnitude by using the techniques and algorithms taught in the course.

Even though this course was meant to be an intensive experience, we had to make compromises to make the project feasible. We decided to restrict the project to building the query processor, but keep updates as simple as possible and completely disregard concurrency control and recovery. We describe our design decisions and our ways in which we have shaped the project in the remainder of this paper.

XIME-P 2006, June 30, Chicago, Illinois

2. THE PROJECT

The system was realized in four succeeding milestones, the first being a purely main-memory-based query engine implementing a small, clean XQuery fragment as the query language. The final milestone is a (moderately) complete XML query processor with efficient secondary storage structures, and algebraic and cost-based query optimization.

Milestone 1: An In-memory XQuery Evaluator

As first milestone, the students built an in-memory query engine for *composition-free XQuery* (XQ) [3]. XQ supports XQuery for-expressions, conditionals, node construction, and (downward) navigation in the input document, but excludes XQuery features such as recursion, duplicate elimination, reordering, and aggregation. The abstract syntax of XQ is shown in Figure 1.

$$
\begin{array}{rcl}
\mathit{query} ::= & & () \mid \langle a \rangle \mathit{query} \langle / a \rangle \mid \mathit{query}\ \mathit{query} \\
& \mid & \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\
& \mid & \text{for } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ return } \mathit{query} \\
& \mid & \text{if } \mathit{cond} \text{ then } \mathit{query} \\
\mathit{cond} ::= & & \mathit{var} = \mathit{var} \mid \mathit{var} = \mathit{string} \mid \text{true}() \\
& \mid & \text{some } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ satisfies } \mathit{cond} \\
& \mid & \mathit{cond} \text{ and } \mathit{cond} \mid \mathit{cond} \text{ or } \mathit{cond} \mid \text{not}(\mathit{cond}) \\
\mathit{axis} ::= & & \text{child} \mid \text{descendant} \\
\nu ::= & & a \mid * \mid \text{text}()
\end{array}
$$

Figure 1: Abstract syntax of XQ.

For the denotational semantics of XQ we refer to the course material online [5]. The semantics is just what one would expect from the XQuery standard, with the following exception. To keep things simple, we made the restriction that students only needed to implement comparisons for the case that variables bind to text nodes. They were allowed to check this at runtime and exit with an error message if two nodes to be compared are not text nodes.

After familiarizing themselves with the XQuery semantics and getting hands-on experience with the Galax XQuery evaluator [4], the students went to work and implemented their own in-memory evaluator for XQ. We provided the students with C^{++} skeleton code consisting of a scanner and a parser for XQ and XML documents.

Teaching Goal. The primary goal was to ensure that the students understood the XQ semantics. Moreover, we wanted to offer enough time for participants to freshen up their C^{++} programming skills. With international masters students in our class who had done their undergraduate degrees at other universities, we could not expect that all participants could build upon the same level of C^{++} programming experience.

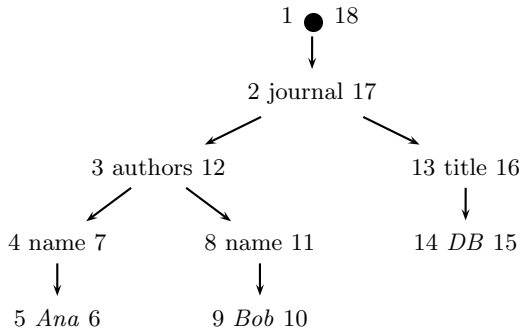


Figure 2: XML document with `in` and `out` labels.

Milestone 2: Accessing Secondary Storage

The task of the second milestone was to write an XQ evaluator that uses secondary storage and does not require building the DOM tree of the input XML document. It can be verified that in XQ, variables are always bound to *single* nodes of the input document [3]. XQ queries can be evaluated without keeping more than just the current bindings of variables to nodes in main memory. The Berkeley DB storage manager was used to keep the data in secondary storage and to fetch only those nodes into main memory that are currently necessary for query evaluation.

Let consider the work involved in this step in more detail. We assume that the nodes in an XML document tree are assigned two numerical values "`in`" and "`out`" in a depth-first left-to-right preorder traversal of the tree (also known as document order). An example of such an assignment is shown in Figure 2, where the `in` values are at the left and the `out` values at the right of node labels.

For storing XML documents, we employ extended access support relations (XASR) [1] with the relational schema

`Node(in, out, parent_in, type, value)`

where each tuple encodes one node of the input XML document, and `in` and `out` are primary keys. For a given node (*) "`in`" is the number of (opening and closing) tags encountered before its opening tag; (*) "`out`" is the number of (opening and closing) tags encountered before its closing tag; (*) "`parent_in`" is the `in` value of its parent node; (*) "`type`" is the type of the node, i.e., `root`, `element`, or `text`; and (*) "`value`" is (1) its label if it is an element node, (2) its text content if it is a text node, (3) or NULL if it is the root.

Example 1. The nodes labeled 'journal' and *Ana* in the XML document of Figure 2 are represented in XASR as the tuples (2, 17, 1, element, journal) and (5, 6, 4, text, *Ana*).□

Obviously, XML documents stored using this schema can be reconstructed, because (1) the child relation is preserved by the `parent_in` values, and (2) the order of the children of a node is preserved by the `in`/`out` values. Further, using this schema, structural joins like `child` or `descendant` can be easily expressed. Consider two nodes x_i and x_{i+1}. Then,

$$x_{i+1} \texttt{ is child of } x_i \Leftrightarrow x_{i+1}.\texttt{parent\_in} = x_i.\texttt{in}$$

$$x_{i+1} \texttt{ is descendant of } x_i \Leftrightarrow x_i.\texttt{in} < x_{i+1}.\texttt{in} \wedge x_i.\texttt{out} > x_{i+1}.\texttt{out}$$

Teaching Goal. In this second phase, the students gathered first experiences using the storage manager and its libraries. At this point, the availability of a mailing list proved useful, as students could help each other with advice regarding the installation and use of the storage manager.

Milestone 3: Algebraic Query Optimization

For the third milestone, an algebraic query optimizer was added to the XQ evaluators. The idea is to rewrite for-loops and certain if-conditions as relational algebra expressions. These subexpressions are optimized using selection-pushing heuristics, i.e. pushing selections as far down as possible using the usual equivalences of relation algebra, and creating joins out of products. Finally, the logical algebra operators need to be implemented by physical operators.

The TPM Algebra. We describe the steps involved more closely. We introduced the *TPM algebra* consisting of projections, selections, cross products, and joins, and further a "super-for-loop" operator called *relfor*. Intuitively, relfor-expressions have the structure

relfor *vartuple* in *xasr-alg* return *expression*

where *xasr-alg* is a relational algebra expression over the XASR relation. Let *vartuple* be the tuple $(\$x_1, \$x_2, \ldots, \$x_k)$, of variables $\$x_1$ to $\$x_k$, where we assume $\$x_1$ bound to the root node (in our XASR encoding always having the `in`-value 1). Then, the result of *xasr-alg* is a k-ary relation.

We define the semantics of an XQ expression α with n free variables using a function $[\![\alpha]\!]_n$ that takes a n-tuple of trees as input (i.e., an *environment* for n variables). On input tree t, query Q evaluates to $[\![Q]\!]_1(t)$.

For a relational algebra expression α, let

$$[\![\alpha]\!]_n(\$x_1 \Rightarrow v_1, \ldots, \$x_n \Rightarrow v_n),$$

be the relation obtained by evaluating α with each occurrence of "external" variable $\$x_i$ in (selection conditions of) α interpreted as constant v_i, and with resulting tuples sorted hierarchically in document order[1]. Let the symbol $\uplus$ denotes list concatenation and $[\ldots]$ the list constructor. Then,

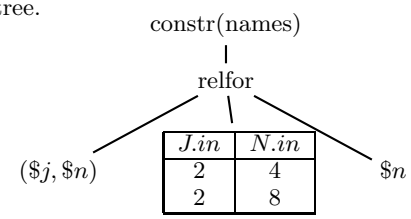
$$[\![\text{relfor } (\$x_{n+1}, \ldots, \$x_{n+k}) \text{ in } \alpha \text{ return } \beta]\!]_n(t_1, \ldots, t_n) :=$$
$$\biguplus \Big[[\![\beta]\!]_{n+k}(t_1, \ldots, t_n, in^{-1}(a_{n+1}), \ldots, in^{-1}(a_{n+k})) \mid \langle a_{n+1}, \ldots, a_{n+k} \rangle \in [\![\alpha]\!]_n(\$x_1 \Rightarrow in(t_1), \ldots, \$x_n \Rightarrow in(t_n)) \Big]$$

where "in" maps nodes to their `in`-values and "in^{-1}" is the inverse mapping an in-value to a given node.

Example 2. For instance, evaluating the query

```
<names> { for $j in /journal return
            for $n in $j//name return $n } </names>
```

on the input document from Figure 2 yields the following operator tree.



The expression at the root is responsible for constructing the XML node with label "names" around the computed result. Below it, there is a relfor-expression that computes the two nested for-loops.

[1] A relation R consisting of tuples with `in`-values of nodes from a document is sorted hierarchically in document order if for all $t_i, t_j \in R$ with $i < j$, there is an attribute A_k such that for all $l < k$, $t_i.A_l = t_j.A_l$ and $t_i.A_k < t_j.A_k$.

Here, the algebraic expression over the XASR relation has already been evaluated. If the physical operators in the implementation are all order-preserving, then the intermediate result is sorted hierarchically in document order. The relfor-expression is evaluated like an imperative for-loop, so the vartuple $(\$j, \$n)$ is successively bound to $(2, 4)$ and $(2, 8)$, representing a pair of `in`-values of nodes in the original document tree. In each binding, the subtree to which variable $\$n$ is bound is written to the output. Consequently, the result nodes are output in document order. We will address the issue of order in more detail later. □

Note that TPM ("the professor's mistake") is not a query algebra in the usual sense. Yet it is sufficient for our purposes and much simpler than existing XQuery algebras. Moreover, we have gracefully reduced the problem of optimizing XQuery to that of optimizing relational algebra queries. All the material about relational query optimization that was presented in the lectures holds immediately[2].

Rewriting For-Loops into TPM. Let a relational algebra expression without union and difference be in *project-select-product normal form* (PSX) if it is of the form

$$\pi_{A_1,\ldots,A_m}(\sigma_{\phi_1\wedge\cdots\wedge\phi_k}(R_1 \times \cdots \times R_n))$$

and the ϕ_i are atomic conditions of the form $A = A'$ or $A = c$, for c a constant. We abbreviate such a query as

$$\text{PSX}((A_1, \ldots, A_m), \phi_1 \wedge \cdots \wedge \phi_k, (R_1, \ldots, R_n)).$$

Then, XQ for-loops can be rewritten into relfor-expressions using rewrite rules such as the following two:

for \$y in \$x/a return $\alpha \vdash$
relfor (\$y) in PSX(R.in, R.parent_in=\$x ∧
R.type=elem ∧ R.value=a, XASR[R]) return α

for \$y in \$x//a return $\alpha \vdash$
relfor (\$y) in
PSX(R_2.in, R_1.in=\$x ∧ R_1.in<R_2.in ∧ R_2.out<R_1.out ∧
R_2.type=elem ∧ R_2.value=a,
(XASR[R_1], XASR[R_2])) return α

Example 3. Applying the previous rewrite rules to the for-loops in the query of Example 2 yields the TPM expression shown in Figure 3. □

Nested relfor-expressions can be merged according to the following rule. Let the names $R_1, \ldots, R_i, S_1, \ldots, S_j$ be pairwise different, then

relfor $(\$x_1, \ldots, \$x_m)$ in
PSX($(A_1, \ldots, A_m)$, ϕ, $(R_1, \ldots, R_i)$) return
relfor $(\$y_1, \ldots, \$y_n)$ in
PSX($(B_1, \ldots, B_n)$, ψ, $(S_1, \ldots, S_j)$) return $\alpha \vdash$

relfor $(\$x_1, \ldots, \$x_m, \$y_1, \ldots, \$y_n)$ in
PSX($(A_1, \ldots, A_m, B_1, \ldots, B_n)$, $\phi \wedge \psi'$,
$(R_1, \ldots, R_i, S_1, \ldots, S_j)$) return α

where ψ' is obtained from ψ by replacing each occurrence of $\$x_i$, for $1 \le i \le m$, by A_i.

Example 4. By merging the relfor-expressions from Example 3, we obtain the TPM expression shown in Figure 4. The PSX subexpression now computes the bindings of variables $\$j$ and $\$n$ to nodes in the document tree.

[2]Close in spirit to TPM's goal, [2] reports on a compilation procedure that derives relational algebra plans from arbitrarily nested XQuery FLWOR expressions.

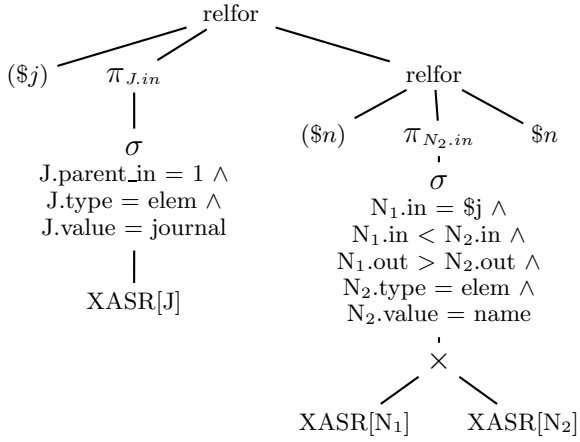


Figure 3: TPM Expression of Example 3.

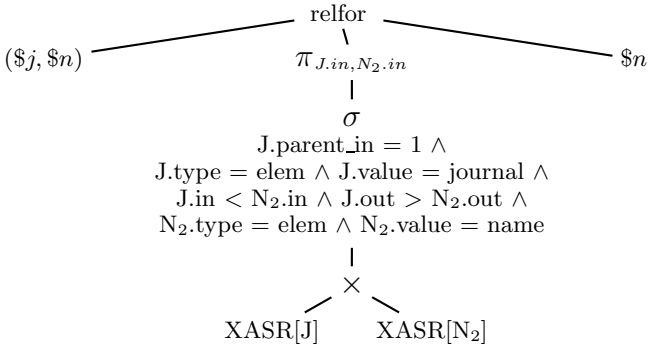


Figure 4: Merged relfor-expression of Example 4.

Note that because N_1.in = \$j = J.in, the relations J and N_1 are the same and we can safely drop N_1. □

To merge relfor-expressions, we need to strictly cohere to the merging rule above. For instance, consider the query from Example 2 modified to have a construction of a `j`-labeled node between the for-loops:

```
<names>{ for $j in /journal return <j>{
          for $n in $j//name return $n }</j> }</names>
```

Then, the relfor-expressions could not be merged offhand. This is because for documents containing `journal`-nodes without children, the construction of *empty* `j`-labeled nodes must still be performed. A merged relfor would always construct *non-empty* `j`-labeled nodes.

As a consequence to this strict merging rule, the evaluation for the above query may be less efficient than for the syntactically very similar query from Example 2:

(1) The relational algebra expression constructed from the inner for-loop will be evaluated for each new binding of $\$j$. One solution to this problem is to extend TPM by left-outer-joins.

(2) Further, when the outer variable ($\$j$ in our example) is bound to the in-value of a node, then only this in-value is available when the relfor-expression for the inner for-loop is evaluated. Yet for computing the descendants of the node bound by $\$j$, we also need the corresponding out-value. Consequently, this value has to be retrieved from the database, which requires an additional join. This overhead can be avoided, e.g., by modifying the vartuples in relfor-expressions so that they also contain the out-value of the bound nodes.

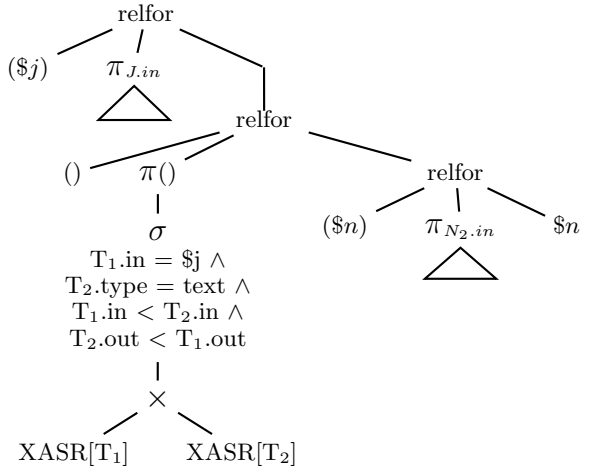


Figure 5: TPM Expression of Example 5. The triangles represent the same subtrees as in Figure 3.

Rewriting If-Expressions into TPM. We only considered if-expressions with the else-part empty and with conditions constructed using "some", "and", and equality tests "A = B" or "A = c", but without "or", "not", or "every". This is because we can only map pass-fail decisions to our TPM fragment of relational algebra. The rules for rewriting into TPM are of the form

$$\text{if } \phi \text{ then } \alpha \text{ else } () \vdash \text{relfor } () \text{ in ALG}(\phi) \text{ return } \alpha$$

where ALG(ϕ) is the rewriting of the condition to TPM. We refer to [5] for the complete set of rules.

We assume that conditions define nullary relations. Note that the empty relation (obtained e.g. by projection $\pi_\emptyset$) admits two possible databases/relations, namely the empty relation ("false") and the nullary relation with the empty tuple ("true"), not just a single relation, the empty one.

EXAMPLE 5. Consider the query

```
<names>{ for $j in /journal return
           if some $t in $j//text() satisfies true())
             then for $n in $j//name return $n
             else () }</names>
```

and its TPM expression with un-merged relfor-expressions in Figure 5. In the second relfor-expression, the projection is on the empty tuple, hence checking whether the condition holds. The three relfor-expressions can further be merged.□

The Role of Order. In the evaluation of a relfor-expression

$$\text{relfor}(\$x_1, \ldots, \$x_k) \text{ in } \alpha \text{ return } \beta$$

with relational algebra expression

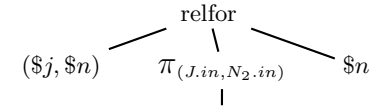
$$\alpha = PSX((A_1, \ldots, A_k), \phi, (R_1, \ldots, R_m))$$

we require that the relation $R[\alpha]$ computed by evaluating α

1. has attributes $A_1, \ldots, A_k$ (in this order),
2. $R[\alpha]$ is sorted hierarchically in document order.

If an XQ query contains no some-expressions, TPM rewriting which strictly adheres to the rules from above will suffice. However, if there are for-loops nested inside if-expressions, then we need to take additional steps to ensure the correct ordering.

For instance, consider the query in Example 5 and the TPM expression in Figure 5. If the translation to TPM were strictly by the rules (and provided all physical operators are order-preserving), we would obtain the following operator tree during evaluation on the document from Figure 2 (the copies N_1 and T_1 of J have been dropped; further, we only show the in-attributes):



J.in	...	*T2.in*	...	*N2.in*	...
2	...	5	...	4	...
2	...	5	...	8	...
2	...	9	...	4	...
2	...	9	...	8	...

While the intermediate result is in document order, when we project on *J.in* and $N_2.in$, we will have to additionally remove duplicates.

We presented three approaches to this ordering problem.

(a) If we sort the tuples in the intermediary relation $R[\alpha]$ accordingly, e.g. by implementing external sorting, we suffer no further restrictions on how to evaluate the relational algebra expression α.

(b) In Example 6 of milestone 4, we discuss a solution to this problem based on pushing projections down to achieve semijoin semantics.

(c) Alternatively, we can ensure that the input for the final projection operator already meets requirements (1) and (2), which also allows us to remove duplicates during projection in one pass.

In pursuing the latter two approaches, we can evaluate XQ without implementing a sorting operator. In the following, we introduce the basic strategy which was implemented in the majority of the student projects.

First, relational algebra expressions are evaluated using only order-preserving operators, e.g. such as nested-loops join but no block-nested-loops join. Second, the join order of the relations $R_1, \ldots, R_m$ in α must ensure that each projection `in`-attribute A_i of α is the attribute of the relation R_i $(1 \leq i \leq k)$.

This yields the desired effect. If all physical operators are order-preserving, then the intermediary result is guaranteed to be sorted in the correct order w.r.t. the attributes on which we project. This then allows us to conveniently project in one pass without having to sort, as can be seen in case of the intermediary result from our previous example.

J.in	...	*N2.in*	...	*T2.in*	...
2	...	4	...	5	...
2	...	4	...	9	...
2	...	8	...	5	...
2	...	8	...	9	...

Teaching Goal. The third milestone required students to extend the query evaluator from milestone 2 to also deal with relational selections, projections, products, and joins. To keep it simple, we allowed the engines to write to disk each intermediate result, and re-read it whenever necessary as the input of a subsequent operation.

The TPM algebra and the rewriting rules were first introduced in the lecture and also practiced on paper as homework before students started implementing them. However, we did not provide the complete set of rewriting rules, just the main rules to communicate the basic idea. So for students to be able to hand in a working milestone 3, they had

to understand the translation of XQ expressions to TPM, and be able to complete the missing rules by themselves.

The decision whether students should preserve document order during query evaluation and get by without sorting, or rather ignore the order and restore it in the end caused much discussion among students.

Unfortunately, the publicly available distribution of BerkeleyDB does not directly support block-based writing, only block-based reading. This made it difficult to have the students implement external sort and block-nested loop-joins properly by the book.

Several students chose to enforce sorted intermediate results by constructing a clustered B-tree index on the input to the projection operator, thus retrieving the results in the proper order. While this is certainly not an elegant solution, we accepted it as a creative workaround, considering that the workload involved with the third milestone was already quite tough for the majority of students.

Milestone 4: Cost-based Query Optimization and Index Structures

For the fourth milestone, students added cost-based query optimization to their evaluators and B^+-tree index structures on the XASR relations. We asked them to think about query plans for common queries and decide which index to cluster to get the best query performance. For the primary index, the attribute `in` was the natural choice.

In addition, our students implemented the physical operators index-based nested-loops join and index-based selection. Also, they were required to store the statistics on the data in separate external storage structures to be able to estimate the costs of their query plans. As a minimum of information, each implementation maintained the selectivity of each of the element node labels occurring in the document, and the average depth of a node in the data tree, as a gross measure for the selectivities of ancestor-descendant joins.

This task was challenging insofar as the formulas for cost-estimates could not simply be taken out of a book. Instead, students had to be able to transfer the ideas about computing estimates in relational databases and apply them to the context of XML processing. We observed both in the tutorials and the mailing list that this sparked their interest and gave rise to different and interesting approaches.

Finally, the students assigned costs to the query plans produced by their systems and chose the cheapest plan according to their estimates.

Choosing good query plans. We next discuss possible query plans produced by students' optimizers.

EXAMPLE 6. Consider an XML document with many authors and few articles that have information on volumes. The example query returns the list of authors of articles that have information on proceedings volume.

```
for $x in //article return
  if (some $v in $x/volume satisfies true())
    then for $y in $x//author return $y else ()
```

We first rewrite the query into TPM expressions as discussed in milestone 3 and create a single relfor from the three relfors of the two for-expressions and of the if-expression. The PSX expression of this relfor refers to three XASR relations, say A for articles, B for authors, and V for volumes.

By simply mirroring the structure of the query and proceeding bottom-up, we create a nonoptimal query plan QP_0, where the relations are joined as $A \bowtie (B \times V)$ with the join condition A.in=V.parent_in $\wedge$ A.in<B.in $\wedge$ B.out<A.out.

We create a better plan QP_1 by splitting the big join condition into two conditions and pushing down one of them. Also, we reorder the joins to $((A \bowtie_a B) \bowtie_b V)$, where a is A.in<B.in$\wedge$ B.out<A.out and b is A.in=V.parent_in. Note that, according to our previous discussion on the correct order of the result of relfor-expressions, QP_1 is order-preserving.

After projection pushing and a second join reordering, we obtain the query plan QP_2 of Figure 6, where the outermost join is not anymore order-preserving. We enforce the right order by pushing the projection π(A.in, A.out) under the outermost join. Note that the innermost join and this projection simulate now a semijoin. The plan QP_2 is clearly better than QP_1, because (1) only those articles that have volumes are checked for authors, (2) the more selective (innermost) join is evaluated first, and (3) both joins are implemented as index nested-loop joins (INL). □

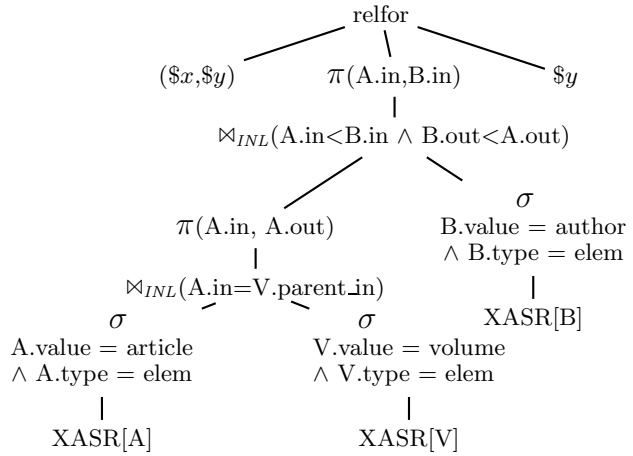


Figure 6: Query plan QP_2 used in Example 6.

Teaching Goal. We wanted our students to realize that the more accurately the rankings of query plans by their cost function are, the better their implementation would perform in the final benchmarks. Calibration of course required them to test their implementation for the same query and alternative query plans, and to take running times to see how rankings by their cost function actually matched reality.

Here, we again enjoyed many discussions with enthusiastic students who competed for the best approach.

Industrious students were rewarded with bonus points if they implemented either pipelining or cost-based join reordering. While the final milestone was perceived by many as particularly tough, especially as some students realized that they had to change their code structure more than anticipated between milestones three and four, it was rewarding to see that nearly everybody who completed the fourth milestone went for the bonus points as well.

At this point in their project, the students had started to identify themselves very much with "their" query engine, and were willing to invest even more to have their engine to become one of the best.

3. THE GRADING SYSTEM

The best grade is represented by 100 points, which could be obtained solely in the final exam. To be admitted to the exam, however, the students had to successfully finish a runnable engine at latest one week prior to the exam. To pass the exam, they also had to achieve a minimum of 50 points in the exam.

Each milestone had allocated approximately four weeks before its early-bird review. In this time, the weekly student meetings were dedicated to discussions and assignments on the topic of that milestone [5]. Besides the early-bird reviews, we offered additional reviews every week, depending on students' request.

A successful submission of a milestone implementation by the early-bird review brought two points. The penalty for missed deadlines (materialized as negative points) increases with the number of weeks of delay. The grading system rewards efficient engines. To support excellence, the 10% and 25% most scalable query engines got additional bonus points. As a result, 25% of the students that successfully passed the exam got more than 100 points in total.

Teams. For the first two milestones we allowed teams of up to two students only. As an experiment, starting with the third milestone, pairs of teams could join forces (for teams of up to four students). We thought this would be beneficial for the migration of knowledge and code. At the same time, small teams completing the final milestones were rewarded a few additional points. We experienced that teams of two were mostly considered optimal.

Grading the Engines. The infrastructure for grading the engines comprised (1) an online submission&test system and (2) milestone reviews, where students answer questions related to the design and implementation of their engines.

(1) The submission&test system is designed to allow students to submit their engines' code via a Web interface at any time and as often as necessary, and to notify the students via email within a half a day on possible problems encountered during the testing phase. Because of all these key features, the system played a irreplaceable role in the development of the XQ engines.

The system was implemented under Linux using Python and Shell scripts and works as follows. The submissions are stored in a submission pool and picked up using a fair scheduling by a tester running on a different machine. To test them, the query engines are recompiled and run under memory and time constraints. Section 4 details on the used testbed. When the tests for an engine are finished, the students is sent an email containing detailed test results, e.g., engine run-time errors, scalability problems if any, the answers to the public queries in case they differ from the correct answers, and the timing.

(2) The milestone reviews were led by teaching assistants, who were informed in advance about the major difficulties encountered by the engines during testing. Then, the discussions were constructively guided towards solving possible engine problems or student misunderstandings. This made it easy to inspect the level of knowledge accumulated by the students and detect software plagiarism.

4. TESTBED FOR THE QUERY ENGINES

Correctness and efficiency tests were used to test the engines. The set of input queries contains public queries (with public answers) as well as secret queries. We did not use benchmark queries proposed in the literature. Some of these benchmark queries are not expressible in XQ, as they involve, e.g., aggregations. Instead we engineered queries that can greatly profit from the optimization techniques treated in the lectures.

The input XML documents we considered are DBLP (250 MB of shallow data), an 16 MB excerpt of DBLP, TREEBANK (80 MB of deeply nested data), and a small handmade document of several kilobytes. All XML documents and public XQ queries are available on the course site [5].

Correctness Tests. For each engine and milestone, the correctness tests used all aforementioned XML documents and up to 16 complex XQ queries. These queries covered fairly all XQ constructs and combinations of them.

Efficiency Tests. After passing the correctness tests, the engines were tested for time and memory scalability. For processing five secret XQ queries on the DBLP document, we allowed only 20 MB of memory and 2 or 30 minutes per query (depending on the running time of our own engine). We chose queries that admit query plans with costs varying by orders of magnitude and that allowed a clear separation of the optimized engines from the unoptimized ones. The queries resemble in spirit the example query used in Section 2 to explain milestone 4.

The Top Five Query Engines. Figure 7 compares the best five engines out of the existing 12. It shows in seconds the user time for each efficiency test, and the total time taken for all efficiency tests. The engines that needed more than 2400 seconds (20 MB) were stopped and assigned 2400 (4800) seconds.

Most of the engines were designed to generate order-preserving plans. Only few of them (not in the top five) decided for B+-trees to keep the temporary (and final) results sorted hierarchically in document order.

The query in the fourth test uses a non-existent node label. This explains the very low timing (under 0.01 seconds) of two engines for this test. Interestingly, the second engine performed very well in all cases but the last. That case corresponds to a query with two nested, yet unrelated, for-loops represented in the query plan by two joins with very different selectivities. Due to unlucky estimates, the second engine decided for an unoptimal query plan (with the very unselective join at the bottom of the plan), which led to the high evaluation time.

Engine	Test 1	Test 2	Test 3	Test 4	Test 5	Total
1	0.11	142.77	28.10	164.95	8.48	344.41
2	0.01	0.01	0.14	0.00	2400	2400.16
3	16.44	175.30	2400	63.76	29.70	2685.20
4	24.72	0.01	2400	0.00	2400	4824.72
5	65.41	163.93	2400	123.66	2400	5153.00

Figure 7: Timing of the Top Five Engines.

5. ACKNOWLEDGMENTS

We thank our teaching assistants Melih Demir, Georgiana Ifrim, Anna Moleda, and Josiane Parreira for their support.

6. REFERENCES

[1] T. Fiebig and G. Moerkotte. "Evaluating Queries on Structure with eXtended Access Support Relations". In *WebDB (Informal Proceedings)*, pages 41–46, 2000.

[2] T. Grust. Purely Relational FLWORs. In *XIME-P*, 2005.

[3] C. Koch. "On the role of composition in XQuery". In *Proc. WebDB*, 2005.

[4] Galax XQuery Engine. http://www.galaxquery.org/.

[5] Saarland University Database Group. Database systems lecture, summer term 2005. http://www-dbs.cs.uni-sb.de/teaching/dbs05/.

[6] S. M. More, T. Pevzner, A. Deutsch, S. B. Baden, and P. Kube. "Building an XQuery interpreter in a compiler construction course". In *Proc. SIGCSE 2005*, pages 2–6, 2005.